IET Information Security

Research Article

IET Journals
The Institution of Engineering and Technology

Back-propagation neural network on Markov chains from system call sequences: a new approach for detecting Android malware with system call sequences

ISSN 1751-8709
Received on 10th December 2014
Revised 29th November 2015
Accepted on 28th December 2015
doi: 10.1049/iet-ifs.2015.0211
www.ietdl.org

Xi Xiao[1], Zhenlong Wang[1], Qing Li[1] ✉, Shutao Xia[1], Yong Jiang[1]

[1]Graduate School at Shenzhen, Tsinghua University, 518055 Shenzhen, People's Republic of China

✉ *E-mail: li.qing@sz.tsinghua.edu.cn*

Abstract: Android has become the most prevalent mobile system, but in the meanwhile malware on this platform is widespread. System call sequences are studied to detect malware. However, malware detection with these approaches relies on common system-call-subsequences. It is not so efficient because it is difficult to decide the appropriate length of the common subsequences. To address this issue, the authors propose a new approach, back-propagation neural network on Markov chains from system call sequences (BMSCS). It treats one system call sequence as a homogeneous stationary Markov chain and applies back-propagation neural network (BPNN) to detect malware by comparing transition probabilities in the chain. Since transition probabilities from one system call to another in malware are significantly different from those in benign applications, BMSCS can efficiently detect malware by capturing the anomaly in state transitions with the help of BPNN. The authors evaluate the performance of BMSCS by experiments with real application samples. The experiment results show that the *F*-score of BMSCS achieves up to 0.982773, which is higher than the other methods in the literature.

1 Introduction

Computer security has always been a serious problem. With mobile terminals becoming more and more prevalent, mobile security is increasingly prominent [1–4]. Due to the growing popularity and openness, Android has attracted the most consideration of malicious elements and a hacker can easily write malicious code and spread it. Malware aiming specifically at Android devices has increased at an alarming rate [5]. Furthermore, Android has unique properties and specific limitations due to its mobile nature. This makes it more difficult to detect malware with conventional techniques. Therefore, it is rather important to develop a new and efficient approach to detecting Android malware.

Researchers have explored two types of methods to detect Android malware. The first type is static analysis, which aims to recognise signatures of the malicious applications without actually executing them [6–9]. Many binary forensic techniques can be used in static analysis, including de-compilation, decryption, pattern matching and so on. Yet these methods cannot detect unknown malware as any application can have distinct signatures by means of encryption and obfuscation [10]. Therefore, the second type of methods, dynamic analysis, is proposed [11–17]. These approaches can monitor application's behaviours such as network access, phone calling and message sending at run time.

The dynamic behaviours of an application are conducted by system call sequences at the end. Therefore, researchers can leverage system call sequences in the dynamic analysis [11–14]. The previous mechanism that uses system call sequences to detect malicious applications usually consists of the following steps: first generating common subsequences of system call sequences of malware, second filtrating the common subsequences appearing in system call sequences of benign applications. If the left common subsequences exist in an application's system call sequence, the application is identified as malware. Nevertheless, these methods are inefficient and cannot achieve a desirable detection rate. The critical limiting factor is the length of the common subsequence. When the common subsequence is too short, the information used to describe the action of an application is insufficient. However, the action is the key character in identifying malicious applications. When the common subsequence is too long, for instance, longer than 45 system calls, it tends to be over fitting [14, 18]. Furthermore, it usually takes too much time to obtain the common system-call-subsequences. The longer the common subsequence is, the more time it takes (even weeks) [18].

To overcome the above shortcoming, in this paper we put forward a new approach for Android malware detection, back-propagation neural network on Markov chains from system call sequences (BMSCS). The Markov chain has been employed in the field of network security [19, 20] and the Markov logic network has been adopted in Android malware detection [21]. Inspired by Xiao *et al.* [22], where they applied homogeneous stationary Markov chains to masquerade detection, we introduce this model in mobile malware detection. Based on the fact that there are some specific correlations between the adjacent system calls (e.g. first memory access, second screen display, then user input requirement), we treat the system call sequence activated by one application as one Markov chain. To get low time complexity, we only take two state dependency into consideration. Each distinct system call corresponds to one unique state in the chain. There are 196 system calls in Android 4.0.4, thus the state number is 196. In [19, 20], the numbers of states in these Markov chains are relatively small. However, there are 196 states in our method. Hence, it cannot solve the problem only by directly analysing each element in the matrices. Some classifiers are required to help further process these matrices.

In our scheme, we first calculate the transition probability matrices by statistical methods and then convert them into vectors of 196×196 dimensions. Our key assumption is that the probabilities of transition from one system call to another are significantly different between malicious applications and benign ones. According to this assumption, the above vectors are fed to the classifier, artificial neural network (ANN), to discriminate malware from benign applications on Android. The classification process consists of the training phase and the detection phase. During the training phase, the ANNs (neural networks, in abbreviation) are trained by back-propagation algorithm, which are called as back-propagation neural networks (BPNNs). Finally, we do the experiments on the malware from [23] and the benign applications downloaded from Google. The results indicate that the *F*-score of our method achieves up to 0.982773, higher than those of [8, 9, 14].

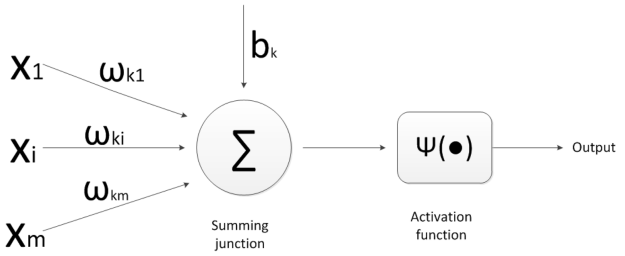


Fig. 1 *Model of a 'neuron' labelled k*

The main contributions are as follows:

i. We use the Markov chains instead of the common subsequences to extract the character of system call sequences. According to the Markov property [22, 24, 25], a system call should only be related with the current system call and should have no relationships with prior calls. The transition probability matrices of the chains only record the frequencies of each pair of system calls. Therefore, BMSCS is independent of the common system-call-subsequences. The time to get the subsequences is eliminated.
ii. We employ BPNNs to capture the anomaly in state transitions in the chain to categorise the Android application as malicious or benign one for the first time based on the assumption that there are remarkable differences between the probabilities of transition from one system call to another in malware and those in benign applications.
iii. We do a large number of experiments on the real dataset including 1227 malicious applications from [23] and 1189 benign ones downloaded from Google. The effects of the depth, the number of hidden-layer nodes and the learning rate of BPNNs on the performance are all investigated. Moreover, we provide the comparison with the previous methods. The high performance of BMSCS justifies our assumption is reasonable.

The rest of this paper is organised as follows. Section 2 offers a concise introduction to system calls, Markov chains and neural networks. The detailed three steps of our new approach are explained in Section 3. Then Section 4 presents the experiment environment and experiment results. The recent literature is briefly surveyed in Section 5 followed by the conclusion and further work in Section 6.

2 Background

2.1 System call

It is the operating system's main task to manage hardware resources and provide a good environment for application developers. To fulfil this task, the kernel provides a series of scheduled kernel functions to application developers by a set of interface, known as system calls. System calls transfer application's requests to the kernel, call the corresponding kernel functions to finish the required work, and then return the results to the application. On the Android platform, the user process cannot directly access hardware devices. When a user process needs to access hardware devices, such as reading disk files and receiving network data, it has to switch from the user mode to the kernel mode by system calls. The Android version in our work is 4.0.4, and there are only 196 system calls in our experiments. The Android kernel is a simplified version of Linux kernel. Compared with traditional Linux, there are some unique behaviours in Android, such as message sending and call making. As a result, there are many differences between the system call actions of Android and Linux.

2.2 Markov chain

A Markov chain is a discrete random process with the Markov property [22, 24, 25]. One process changes randomly, and it is generally impossible to predict with certainty the state of a Markov chain at a given point in the future. However, the statistical properties of the process's future can be predicted [20, 24, 25]. The Markov chain is a good and versatile model and widely used.

Definition 1: A discrete random process $\{c_n, n \geq 1\}$ is a Markov chain with state space $\Theta = \{1, 2, \ldots\}$ if it satisfies the Markov property, i.e. for all $n \geq 1$ and $\theta_m \in \Theta$ with $1 \leq m \leq n+1$

$$\Pr(c_{n+1} = \theta_{n+1} \mid c_n = \theta_n, c_{n-1} = \theta_{n-1}, \ldots, c_1 = \theta_1) = \Pr(c_{n+1} = \theta_{n+1} \mid c_n = \theta_n) \tag{1}$$

That is to say that in the Markov chain the next state depends only on the current state and not on the states that preceded it. The Markov property is very useful in terms of explicitly finding the probability of a vast number of interesting events [26].

Definition 2: A Markov chain $\{c_n, n \geq 1\}$ is homogeneous if the conditional probabilities $p_{ij}(n) = \Pr(c_{n+1} = j \mid c_n = i)$ are independent of n for all $i, j \in \Theta$.

Definition 3: Let $\{c_n, n \geq 1\}$ be a homogeneous Markov chain and

$$p_{ij} = \Pr(c_{n+1} = j, c_n = i), \quad a_i = \Pr(c_1 = i) \quad \text{for all } i, j \in \Theta.$$

$\boldsymbol{P} = [p_{ij}]$ is the matrix of the conditional probabilitiesp_{ij}, called the (one-step) transition (probability) matrix. $\boldsymbol{A} = [a_i]$ is a row vector representing the probability mass function of c_1, called the initial (probability) distribution of the Markov chain.

Theorem 1: A Markov chain $\{c_n, n \geq 1\}$ is completely characterised by the initial distribution $\boldsymbol{A}$ and the transition matrix $\boldsymbol{P}$, i.e.

$$\Pr(c_1 = \theta_1, \ldots, c_{n-1} = \theta_{n-1}, c_n = \theta_n) = a_{\theta_1} \times p_{\theta_1, \theta_2} \times \cdots \times p_{\theta_{n-1}, \theta_n} \tag{2}$$

Definition 4: A Markov chain $\{c_n, n \geq 1\}$ is stationary if the initial distribution $\boldsymbol{A}$ and the transition matrix $\boldsymbol{P}$ satisfy

$$\boldsymbol{A} = \boldsymbol{A} \times \boldsymbol{P} \tag{3}$$

Theorem 2: Let $\{c_n, n \geq 1\}$ be a stationary Markov chain, then

$$\Pr(c_n = i) = \Pr(c_1 = i) = a_i \tag{4}$$

From Theorems 1 and 2, it is easy to obtain:

Theorem 3: Let $\{c_n, n \geq 1\}$ be a homogeneous stationary Markov chain, then

$$\Pr(c_m = \theta_m, \ldots, c_{n-1} = \theta_{n-1}, c_n = \theta_n) = a_{\theta_m} \times p_{\theta_m, \theta_{m+1}} \times \cdots \times p_{\theta_{n-1}, \theta_n}, 1 \leq m < n \tag{5}$$

2.3 Neural network

Inspired by an animal's central nervous system, artificial neural networks (ANNs) attempt to parallel and simulate the functionality and decision-making processes of the human brain. A neural network is a massively parallel distributed processor consisted of simple processing units. The processor has a natural propensity for storing experiential knowledge, making it available for use [27, 28].

ANNs are generally referred to as mathematical models of theorised mind and brain activity. They are presented as systems of interconnected 'neurons' which can compute values from inputs. Fig. 1 shows the model of a 'neuron' labelled k.

The neuron k in Fig. 1 can be depicted by the following three mathematical equations:

Algorithm of Calculating the Transition Probability Matrix

Input: the system call sequence $(c_1, c_2, ..., c_M)$

Output: the transition probability matrix $P = [p_{ij}]_{196\times196}$

$A = [a_{ij}]_{196\times196}$ //declare a temporary matrix

$P = [p_{ij}]_{196\times196}$ //declare a matrix to save the result

```
for (i=0; i<=195; i++) {
        for (j=0; j<=195; j++)
            a_ij = 0;          //initialize the matrix
}
for (i=2; i<=M; i++) {
        x = c_{i-1};
        y = c_i;
        a_xy = a_xy + 1;       //count transition times from one state to another state
}
for (i=0; i<=195; i++){
        sum = 0;
        for (j=0; j<=195; j++)          //calculate transition times from one state to all the other
            sum = sum + a_ij;           // states
        for (j=0; j<=195; j++)
            p_ij = a_ij ÷ sum;
}
```

Fig. 2 *Algorithm of calculating the transition probability matrix*

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{6}$$

$$v_k = u_k + b_k \tag{7}$$

$$y_k = \psi(v_k) \tag{8}$$

$$\psi(v) = \frac{1}{1 + \exp(-av)} \tag{9}$$

BPNN is the most common ANN. It is trained by the back-propagation algorithm, which is a solution to the problem of training multi-layer perceptrons [29]. BPNNs have been applied in many applications such as automotive, robotics, banking and electronic.

3 Our approach

There have been some works that utilise common subsequences of system call sequences [14, 18]. The disadvantage is that when the common subsequence is too short, the information is insufficient to detect malware and while the common subsequence is too long, it may cause over-fitting. Not only that, it takes too much time to get long common subsequences. It is hard to choose the appropriate length of the common subsequences. In this paper, we use the Markov chains instead of the common subsequences to extract the character of system call sequences. BMSCS is independent of the common system-call-subsequences, without extracting the subsequences. Our method can be divided into three steps: (i) character extraction, (ii) training, and (iii) detection.

3.1 Character extraction

On the Android platform, *strace*, a command-line tool, can be used to track and record the system call sequence invoked by a process. Another command, *monkey*, can be used to send the stream of random- and pseudo-user events to applications. We at first use *monkey* to simulate 1000 user events and then employ *strace* to record the system call sequence of one application. Finally, by numbering all the 196 system calls from 0 to 195, we convert the system call sequence from the character form to the number form.

For the sake of simplicity, homogeneous stationary Markov chains are used to extract the character from system call sequences. Each distinct system call corresponds to one unique state of the Markov chain. Thus, we can count the times of transition from one system call to another one to calculate the transition probability matrix. For instance, suppose there are only four system calls, noted as *A*, *B*, *C* and *D*, and the system call sequence is as follows:

$$A-A-B-C-D-B-C-D-A-B-A-C-D-B-C-C$$

We number the system calls *A*, *B*, *C*, *D* as 0, 1, 2, 3. Then the above system call sequence turns into the following form:

$$0-0-1-2-3-1-2-3-0-1-0-2-3-1-2-2$$

The transition times from system call *i* to system call *j* is noted as a_{ij}. In this case, $a_{00} = 1$, $a_{01} = 2$, $a_{02} = 1$, $a_{03} = 0$ and so on. Then we define, p_{ij}, the transition probability from system call *i* to system call *j*, in the following way:

$$p_{ij} = \frac{a_{ij}}{\sum_j a_{ij}} \tag{10}$$

According to formula (10), we can get the transition probability matrix

$$\boldsymbol{P} = (p_{ij})_{4\times4} = \begin{pmatrix} 0.25 & 0.5 & 0.25 & 0 \\ 0.25 & 0 & 0.75 & 0 \\ 0 & 0 & 0.25 & 0.75 \\ 0.33 & 0.67 & 0 & 0 \end{pmatrix}$$

The pseudo-code of calculating the transition probability matrix is shown in Fig. 2

3.2 Training

In our approach, we employ a special network, BPNNs. After calculating all the transition probability matrices of the applications in the training data, all the rows of one matrix are joined head to tail together to construct a vector. The input of the training phase is $\{x(n), d(n)\}$, where $x(n)$ is the vector constructed from the nth sample in the training data and $d(n)$ is the corresponding label. $d(n)$ is defined as follows: when the application is malicious, $d(n) = 1$, otherwise, $d(n) = 0$. In the training step, the primary work is to train the networks so that they have the ability to distinguish malware from benign applications. Note there are 196×196 input-layer nodes and only one output-layer node in our networks. We use the following back-propagation algorithm in Algorithm 1 to train the BPNNs [27, 29]:

Algorithm 1: Algorithm of back propagation:
The algorithm cycles through the training sample $\{(x(n), d(n))\}_{n=1}^{N}$ as follows:

i. **Initialisation**. Pick the synaptic weights and thresholds from a uniform distribution whose mean is zero.
ii. **Presentations of Training Examples**. Present the network an epoch of training examples. For each example in the training data do the forward computation in point (iii) and the backward computation in point (iv).
iii. **Forward Computation**. Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer. The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is defined as

$$v_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n) \tag{11}$$

where $w_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer $l-1$ and $y_i^{(l-1)}(n)$ is the output of neuron i in layer $l-1$ at iteration n. When $i = 0$, $w_{j0}^{(l)}(n) = b_j^{(l)}(n)$ is the bias applied to neuron j in layer l and $y_0^{(l-1)}(n) = +1$. The output of neuron j in layer l is

$$y_j^{(l)}(n) = \psi(v_j(n)) \tag{12}$$

When $l = 1$, $y_j^{(0)}(n) =$ the jth element of $x(n)$. The depth of the network is denoted by L. When $l = L$, calculate the error signal as follows:

$$e(n) = d(n) - y_1^{(L)}(n) \tag{13}$$

iv. **Backward Computation**. Compute the local gradients, δs, of the network by the formula

$$\delta^{(L)}(n) = e(n)\psi'(v_1^{(L)}(n)) \tag{14}$$

for the neural in output-layer L and

$$\delta_j^{(l)}(n) = \psi'\left(v_j^{(l)}(n)\right)\sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) \tag{15}$$

for neural j in hidden-layer l. Then adjust the synaptic weights of the network in layer l using the rule

$$\begin{aligned} w_{ji}^{(l)}(n+1) &= w_{ji}^{(l)}(n) + \alpha\left[\Delta w_{ji}^{(l)}(n-1)\right] \\ &\quad + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n) \end{aligned} \tag{16}$$

where η is the learning-rate parameter and α is the momentum constant.
v. **Iteration**. Iterate the forward computation in point (iii) and the backward computation in point (iv) by presenting new epochs of training examples to the network until the chosen stopping criterion is met.

3.3 Detection

By means of the back-propagation algorithm, the BPNNs have gained the ability to classify the application as benign or malicious. In this step, the network only requires forward computations. First, the transition probability matrices of the applications in the testing data are calculated, and then transformed into the corresponding vectors. At last, we feed the vectors to the trained BPNNs. According to formulas (11) and (12), the final results are obtained by computing the output of each node in the network layer by layer. The value of the output node is between 0 and 1. The networks use this value to do the classification. When the value is >0.5, the application will be recognised as malicious one, otherwise will be recognised as benign one.

4 Experiments and results

4.1 Datasets and experimental design

There are 1227 malicious applications and 1189 benign ones in our experimental datasets. The malware, consisting of 49 malware families, is offered by Zhou and Jiang [23]. By modifying the Chrome Apk-downloader plugin, we downloaded 1189 applications from the Google Play as the benign applications. Half of the malicious application dataset and half of the benign application dataset are used for training and the rest for detection. We wrote the programs of ANNs in C++ and executed them on Ubuntu. In our experiments, the momentum constant of networks, α, is set to zero. To speed up the execution, the programs utilised an application program interface (API), OpenMp, which supports multiprocessing programming.

Let TP denote the number of malicious applications that are correctly detected, and FP refer to the number of benign applications that are falsely detected as malware. On the contrary, TN represents the number of benign applications that are correctly detected, and FN refers to the number of malicious applications that are falsely detected as benign ones. Based on these numbers, some criteria are proposed to evaluate the experiment results, such as true positive rate (TPR), false positive rate (FPR), precision and F-score. TPR, i.e. detection rate, is defined as

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{17}$$

FPR is defined as

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{18}$$

Precision is defined as

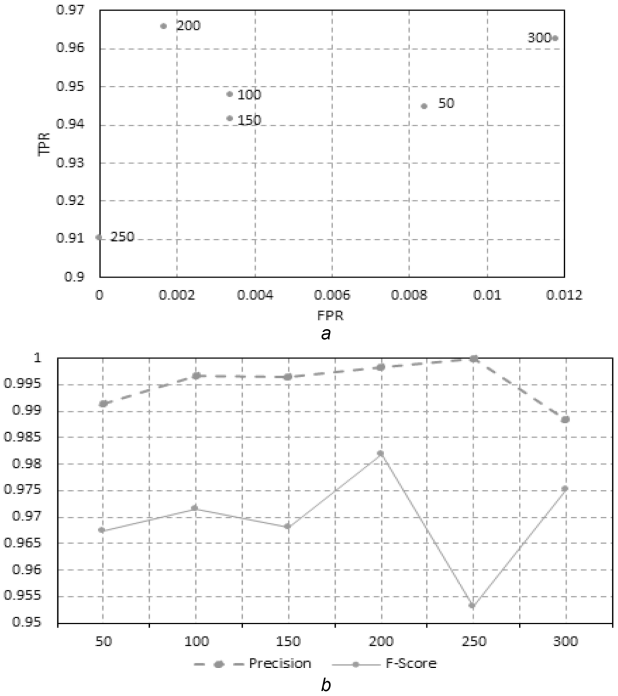
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{19}$$

F-score is a composite evaluation criterion of TPR and FPR, which can indicate the performance of a detection system. When the value of TPR is higher and the value of FPR becomes smaller, the detection system will achieve a higher F-score and a better performance. F-score is the harmonic mean of the precision and the TPR, defined as

$$F\text{-score} = \frac{2 \times \text{Precision} \times \text{TPR}}{\text{Precision} + \text{TPR}} \tag{20}$$

4.2 Experimental analysis

The structure of the networks, including two aspects, i.e. the depth and the hidden-layer node number, impacts the experiment results. However, the determination of the optimal number of hidden layers and that of nodes in each layer is one of the most critical tasks in the ANN design. One starts with no prior knowledge as to the number and size of hidden layers [28]. Thus, in order to get higher F-score and TPR as well as lower FPR, we need to choose the appropriate structure with experimentation.



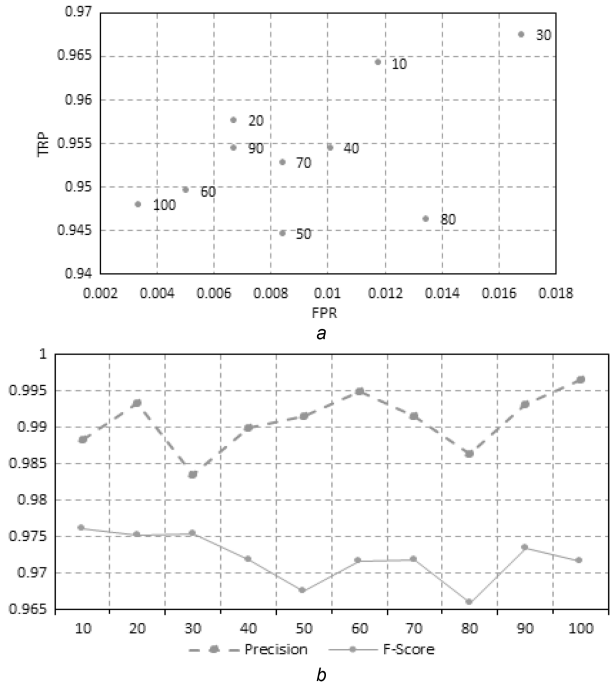
a

b

Fig. 3 *Experiment results of the three-layer networks where the number of hidden-layer nodes ranges from 50 to 300 with the interval of 50*

a Result of TPR and FPR

b Result of *F*-score and precision



a

b

Fig. 4 *Experiment results of the three-layer networks where the number of hidden-layer nodes ranges from 10 to 100 with the interval of 10*

a Result of TPR and FPR

b Result of *F*-score and precision

4.2.1 Three-layer networks: In the case of the three-layer networks, the number of input-layer nodes is 38,416 and that of output-layer nodes is 1. We set the learning rate to 0.9. To build a three-layer network with a good *F*-score, this work has to adjust the number of the hidden-layer nodes to an appropriate value. For the purpose of studying the fine-grain detection performance of the hidden-layer node number, the increment of the hidden-layer node number becomes smaller and smaller as shown in Figs. 3–5, i.e. 50, 10 and 1, respectively.

Fig. 3*a* illustrates the TPR and the FPR of the three-layer networks, of which the hidden-layer node number ranges from 50 to 300 with the interval of 50. The number on the right side of one point is the number of the hidden-layer nodes corresponding to this point. Fig. 3*b* shows the *F*-score and the precision of the networks. As depicted in Fig. 3*b*, when the hidden layer has 200 nodes, the network gets the highest *F*-score of 0.981788. Meanwhile, the FPR is 0.0016835 and the TPR goes up to 0.965798 as shown in Fig. 3*a*. There is no regular pattern between the *F*-score and the number of hidden-layer nodes in Fig. 3*b*. It is because that there is no rule about the performance and the structure of ANNs [28]. Nevertheless, all the *F*-scores are above 0.95, which indicates BMSCS achieves a high performance.

Fig. 4*a* shows the TPR and the FPR of the three-layer networks where the hidden-layer node number ranges from 10 to 100 with the increment of 10. Fig. 4*b* illustrates the *F*-score and the precision of the networks. As plotted in Fig. 4*a*, the network of ten hidden-layer nodes detects 96.4169% malicious applications at a FPR of 0.0117845. Subsequently, it achieves the highest *F*-score of 0.976092 in Fig. 4*b*. It is clear that the network gets a better performance when the hidden-layer node number is <40. If the network has too many hidden-layer nodes, it will follow the noise in the data due to over parameterisation leading to poor generalisation. Therefore, it gets non-optimal results. Moreover, with the increasing number of hidden-layer nodes, training becomes excessively time consuming [28]. Hence, in practice, the hidden-layer node number should be set to 10.

Fig. 5*a* depicts the TPR and the FPR of the three-layer networks, of which the hidden-layer node number ranges from 1 to 40 with the interval of 1. In our experiment, the FP fluctuates within a small range, 3–12, and the value of FP + TN is a constant number of 594. Hence, according to formula (12), the variation of the FPR only depends on the fluctuation of the FP. As the FP could only be ten different values, the FPR could just achieve ten different values. Thus, many points have the same FPR. As shown in Fig. 5*a*, these points are on the same vertical line. Fig. 5*b* shows the *F*-score and the precision of the networks. It can be seen that all the *F*-scores of our method are around 0.975. As illustrated in Fig. 5, when the hidden layer has 37 nodes, the network gets the highest *F*-score of 0.980328 at a TPR of 0.973941 and a FPR of 0.013468.

4.2.2 Four-layer networks: In the case of the four-layer networks, the number of input-layer nodes, the number of output-layer nodes and the value of the learning rate are the same as those in the third-layer network, i.e. 38,416, 1 and 0.9 respectively. The number of second-hidden-layer nodes is kept as a constant of 10. We adjust the first-hidden-layer node number to obtain different detection rates. Fig. 6*a* illustrates the TPR and the FPR of the four-layer networks, of which the first-hidden-layer node number ranges from 50 to 700 with the interval of 50. The number on the right side of one point is the first-hidden-layer node number corresponding to this point. Fig. 6*b* depicts the *F*-score and the precision of the networks. The network of 600 first-hidden-layer nodes arrives at the highest *F*-score of 0.981997. As shown in Fig. 6*a*, simultaneously the TPR goes up to 0.977199 and the FPR goes down to 0.013468. It is obvious that the network has a better performance when the first-hidden-layer node number is between 450 and 650.

To further investigate the fine-grain performance of the first-hidden-layer node number between 450 and 650, Fig. 7*a* demonstrates the TPR and the FPR of the four-layer networks where the first-hidden-layer node number ranges from 450 to 600 with the interval of 10. Fig. 7*b* plots the *F*-score and the precision of the networks. As depicted in Fig. 7*b*, the network achieves the highest *F*-score of 0.982773 while the first-hidden-layer has 490 nodes. At the same time, the TPR is 0.97557 and the FPR is 0.010101. There is little difference in the results of the node number between 490 and 600. The reason may be that the interval, 10, is so small compared with the first-hidden-layer node number that it can be even ignored.

4.2.3 Comparison between three-layer networks and four-layer networks: Table 1 lists the highest *F*-scores of the three-layer networks and the four-layer networks. In general, the more

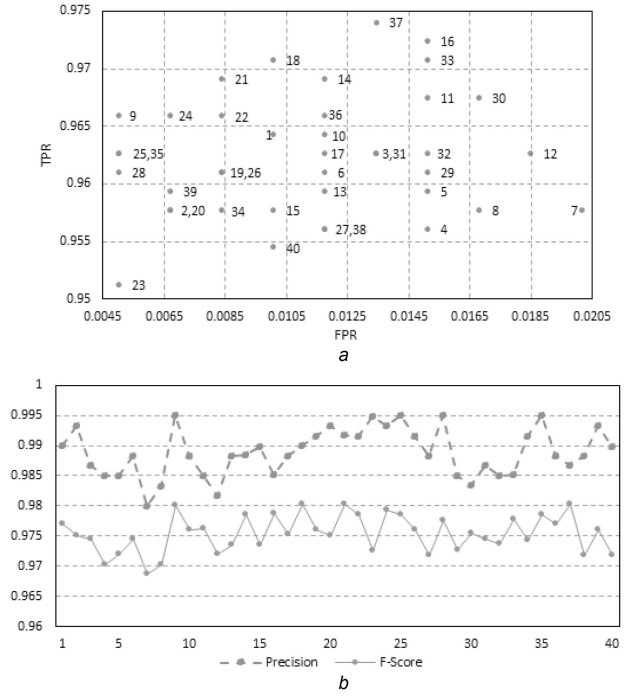


Fig. 5 *Experiment results of the three-layer networks where the number of hidden-layer nodes ranges from 1 to 40 with the interval of 1*
a Result of TPR and FPR
b Result of *F*-score and precision

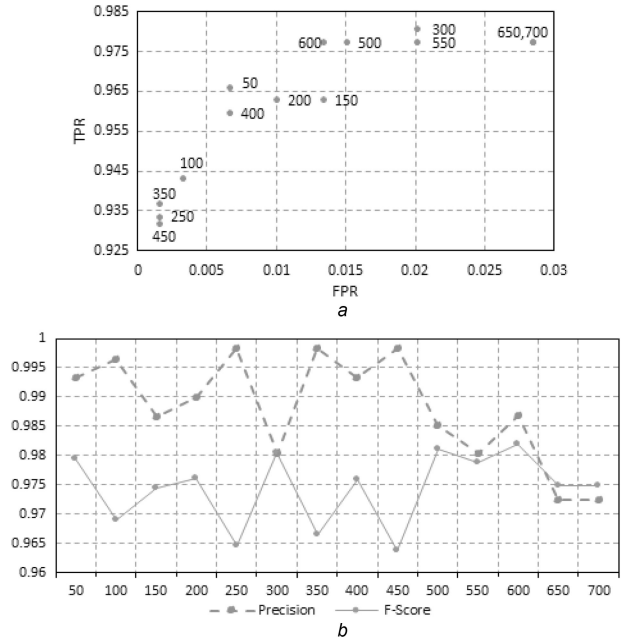


Fig. 6 *Experiment results of the four-layer networks where the number of first-hidden-layer nodes ranges from 50 to 700 with the interval of 50*
a Result of TPR and FPR
b Result of *F*-score and precision

layers the network has, the higher detection rate it will get. Therefore, in the table the *F*-score of the four-layer network is higher than the three-layer network. However, with the more layers, the structure of ANN becomes more complex and it requires more time to train and test. Hence, in the practical use of BMSCS, the tradeoff between the detection rate and the time complexity should be considered in the determination of the network's depth.

4.2.4 Influence of learning rates: Table 2 shows the variation of the training times and the *F*-score with the different learning rates from 0.1 to 0.9, in which the three-layer networks of ten hidden-layer nodes are used. In the cases with the learning rate from 0.9 to 0.2, the network converges. When the learning rate becomes smaller, it needs more times to train the network to convergence. While the learning rate is 0.1, the network cannot converge, even though the training times more than 13,000. It can be seen from the table that the changes of *F*-score are relatively slight with the variation of learning rates. The learning rate mainly decides the learning speed of neural networks. The more training times mean the higher computation complexity. In this case, taking the training times and the *F*-score into comprehensive consideration, we can set the learning rate to 0.7.

4.3 Comparison with other methods

To evaluate the experiment results, this work compares the *F*-score, the TPR and the FPR with Peiravian [8], Drebin [9] and system call sequence droid (SCSDroid) [14]. The results are shown in Table 3. The method in [8] employed static analysis to detect malware by combining permissions and API calls as the input of machine learning methods. Drebin [9] performed a broad static analysis on permissions, API calls and network addresses and used support vector machine (SVM) to do the classification. SCSDroid [14] used common system-call-subsequences to identify malicious repackaged applications. The *F*-scores of these approaches were not explicitly given in [8, 9, 14]. We calculate them by using formulas (17)–(20) according to the results in these papers.

Although the FPRs of Drebin and BMSCS are the same, the TPR of BMSCS is much higher than that of Drebin. BMSCS exhibits good performance in terms of a much higher *F*-score. Compared with Peiravian, the TPR of BMSCS is much higher and the FPR is much lower, and then BMSCS achieves a much higher *F*-score. With comparison to SCSDroid, the TPR of BMSCS is a little lower, but the FPR is much lower and consequently the *F*-score is much higher. An *F*-score close to 1 indicates the good performance on correctly recognising malware. It can be concluded that, in general, BMSCS performs better than Drebin, Peiravian and SCSDroid. This can attribute to the following reasons. At first, system calls can satisfactorily describe the dynamic behaviours of one application, which outperforms the static analysis of Drebin and Peiravian. Second, the relationships between two adjacent system calls are taken into consideration in the Markov chain. Furthermore, ANNs have the strong self-study ability and the potential for high fault tolerance. Thus, BMSCS can get a better detection result.

5 Related works

There are two types of methods with respect to Android malware detection, namely static/dynamic analysis. In static analysis, it does not need to execute an application but to check whether an application has malicious signatures [6–8, 30, 31]. Enck *et al.* [6] treated violating permissions that an application claims at install time as signatures. However, the violating permissions are neither necessary nor sufficient. Thus, the method did not get a high detection rate. Meanwhile, Fuchs *et al.* [7] disassembled the source file and checked permissions and data flow of application components in source code. This approach also suffers from the drawback of Enck *et al.* Zhou *et al.* [31] proposed a method to detect the repackaged malware. It calculates similarity scores between the original app and the repackaged malware. Nevertheless, it is difficult to get original official applications especially for those unpopular ones. In [8], Peiravian and Zhu combined permissions and API calls as features and used machine learning methods to recognise malicious applications. Drebin [9] employed SVM with the comprehensive features involving permissions, API calls and network addresses. Aafer *et al.* [32] extracted relevant features to malware behaviour captured at API level, and evaluated different classifiers using the generated feature set.

All the static analysis methods cannot detect malware that has unknown malicious signatures. Therefore, the dynamic analysis methods are put forward. In dynamic analysis, it requires running information of applications [11–13, 15, 33]. Enck *et al.* [15] proposed TaintDroid, an extension to the Android, which tracks the flow of privacy sensitive data through 30-party applications.

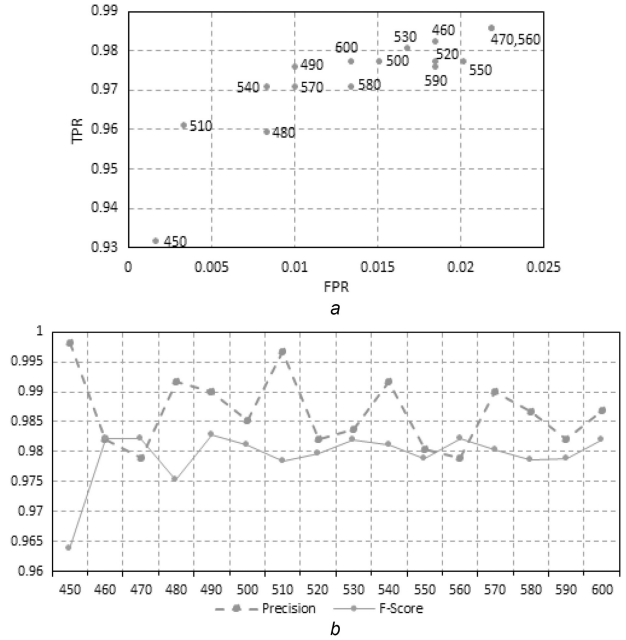


Fig. 7 *Experiment results of the four-layer networks where the number of first-hidden-layer nodes ranges from 450 to 600 with the interval of 10*
a Result of TPR and FPR
b Result of *F*-score and precision

TaintDroid can only detect the specific malware attempting to steal sensitive data. In [11], Blasing *et al.* presented AASandbox which is able to perform both static and dynamic analyses to automatically detect suspicious applications. AASandbox gets a low detection rate. In [12], Burguera *et al.* put forward CrowDroid, which counts the numbers of system calls, and has the same defect as AASandbox.

Another dynamic analysis mechanism is to use common subsequences of system call sequences. Rozenberg *et al.* [18] employed common system-call-subsequences to detect malware on the Windows platform. In the training phase, the method utilises sequential pattern discovery using equivalence classes (SPADE) and genetic algorithm to extract common subsequences of system call sequences that only exist in malware but not in benign software. In the testing phase, it monitors the software in run time and checks whether there exists a match between a portion of the sequences of the run-time system calls and one or more of the common subsequences. In [14], Lin *et al.* proposed SCSDroid, which adopts the thread-grained system call sequence activated by applications on the Android platform. Like the way in [18], SCSDroid extracts the malicious common subsequences from system call sequences of malware. These methods all suffer from the same drawback: the time complexity of the algorithm to obtain the common subsequences is too high, even taking several weeks. In our method, instead of the common system-call-subsequences, Markov chains are used to describe the dynamic character. The identification does not depend on the common subsequences, which avoids the critical issue of the length of the subsequence. Our method eliminates the time of getting the common subsequences and simultaneously improves the detection performance.

The Markov theory has been applied in the field of network security [19, 20]. Gupta and Dharmaraja [19] presented a general analytical framework of dependability model for a voice-over Internet protocol (VoIP) system. This model is analysed with the semi-Markov process which captures the effects of non-Markovian nature of the time spent at various states of the system. Kuang *et al.* [20] proposed a fuzzy prediction method of network security situation. The method adopts the transition matrix of the Markov to depict the correlation of network security and predict the security status. In [19, 20], the state numbers in the Markov chains are relatively small. Therefore, each element in the transition matrices can be analysed directly by researchers. However, the transition matrices in our method are 196×196 dimensions. The number of elements in the matrix is so great that we cannot analyse every element by ourselves. In our method, the elements are further processed by ANNs. Owing to the strong self-study ability and the potential for high fault tolerance of ANNs, BMSCS gets a good detection result.

In [22], Xiao *et al.* assumed that shell command sequence can be regarded as a Markov chain and employed Markov chains for masquerade detection. Inspired by Xiao *et al.* [22], this paper assumes the system call sequence activated by an application as a homogeneous stationary Markov chain. Taking the correlations between the system calls into consideration, BMSCS achieves an *F*-score as high as 0.982773.

Table 1 Comparison between two networks

Network	*F*-score	TPR	FPR
three-layer network	0.981788	0.965798	0.0016835
four-layer network	0.982773	0.975570	0.0101010

6 Conclusion and further work

This work proposes a new method, BMSCS, which adopts BPNNs to classify the transition probability matrix of the Markov chain generated from system call sequences for Android malware detection. We assume that the transition probabilities from one system call to another one are significantly different between malicious applications and benign ones. The *F*-score of BMSCS can reach 0.982773, higher than the other methods, which verifies the rationality of our assumption.

There are some directions for further study based on Markov chains from system call sequences to detect mobile malware. At first, two system call dependency, i.e. the next system call only dependent on the current one, is considered in our Markov chain with low time complexity. In the future, we will implement the method on the mobile phones with limited resources. Second, some other classifiers besides ANNs can be used. It is important to find which classifier is the best in this problem. Third, the number of the states of Markov chains can be reduced, and based on the small matrices it will be easy to identify malware. In this case, how to reduce the state number becomes challenging. In addition, there are some limitations of our approach. If the attacker writes the malware to issue a lot of normal system call sequences like a benign application, our method cannot detect this malware. Our future work will focus on how to detect this kind of malware.

Table 2 Influence of learning rates

Learning rate	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
training times	—	683	128	64	44	42	41	40	41
F-score	—	0.97738	0.97724	0.97549	0.97533	0.97613	0.97613	0.97525	0.97609

Table 3 Comparison with other methods

	F-score	TPR	FPR
Drebin [9]	0.8694	0.94	0.01
Peiravian [8]	0.9525	0.948	0.0312
SCSDroid [14]	0.969697	0.979592	0.02
BMSCS	0.982773	0.975570	0.01

7 Acknowledgments

This work is supported by the NSFC projects (61202358, 61402255,61371078), the National Basic Research Program of China (2012CB315803), the National High-tech R&D Program of

China (2015AA016102, 2014ZX03002004), the Research Fund for the Doctoral Program of Higher Education of China (20130002110051) and the RD Program of Shenzhen (JCYJ20150630170146831, ZDSYS20140509172959989, JSGG20150512162853495, Shenfagai[2015]986).

8 References

[1] Yerima, S.Y., Sezer, S., McWilliams, G.: 'Analysis of Bayesian classification-based approaches for Android malware detection', *IET Inf. Sec.*, 2014, **8**, (1), pp. 25–36

[2] Liu, J., Pan, W., Hu, J., *et al.*: 'Research of secure ecosystem based on Android platform'. Proc. of Int. Conf. on IET Cyberspace Technology (CCT 2013), Beijing, China, November 2013, pp. 376–380

[3] Zhao, X., Fang, J., Wang, X.: 'An Android malware detection based on permissions'. Proc. of Int. Conf. on Information and Communications Technologies (ICT 2014), Nanjing, China, May 2014, pp. 1–5

[4] Xiao, X., Xiao, X., Jiang, Y., *et al.*: 'Detecting mobile malware with TMSVM'. Proc. of 10th Int. Conf. on Security and Privacy in Communication Networks (SecureComm 2014), Beijing, China, September 2014

[5] Juniper Networks Mobile Threat Center: 'Third annual mobile threats report: March 2012 through March 2013'. Available at http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf

[6] Enck, W., Ongtang, M., McDaniel, P.: 'On lightweight mobile phone application certification'. Proc. of 16th ACM Int. Conf. on Computer and Communications Security, New York, USA, November 2009, pp. 235–245

[7] Fuchs, A.P., Chaudhuri, A., Foster, J.S.: 'SCanDroid: automated security certification of Android applications'. Proc. of 31st IEEE Int. Conf. on Security and Privacy (S&P 2009), California, USA, 2009

[8] Peiravian, N., Zhu, X.: 'Machine learning for android malware detection using permission and API calls'. Proc. of 25th IEEE Int. Conf. on Tools with Artificial Intelligence, Herndon, USA, November 2013, pp. 300–305

[9] Arp, D., Spreitzenbarth, M., Hubner, M., *et al.*: 'Drebin: effective and explainable detection of android malware in your pocket'. Proc. of Int. Conf. on Network and Distributed System Security Symp. (NDSS 2014), San Diego, California, USA, February 2014

[10] Moser, A., Kruegel, C., Kirda, E.: 'Limits of static analysis for malware detection'. Proc. of 23th Annual Computer Security Applications Conf. (ACSAC), Florida, USA, December 2007, pp. 421–430

[11] Blasing, T., Batyuk, L., Schmidt, A.D., *et al.*: 'An android application sandbox system for suspicious software detection'. Proc. of 5th IEEE Int. Conf. on Malicious and Unwanted Software, Lorraine, France, October 2010, pp. 55–62

[12] Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: 'Crowdroid: behavior-based malware detection system for android'. Proc. of 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, New York, USA, 2011, pp. 15–26

[13] Isohara, T., Takemori, K., Kubota, A.: 'Kernel-based behavior analysis for android malware detection'. Proc. of 7th Int. Conf. on Computational Intelligence and Security, Hainan, China, December 2011, pp. 1011–1015

[14] Lin, Y.D., Lai, Y.C., Chen, C.H., *et al.*: 'Identifying android malicious repackaged applications by thread-grained system call sequences', *Comput. Secur.*, 2013, **39**, pp. 340–350

[15] Enck, W., Gilbert, P., Chun, B.G., *et al.*: 'TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones', *Commun. ACM*, 2014, **57**, (3), pp. 99–106

[16] Shabtai, A., Kanonov, U., Elovici, Y., *et al.*: 'Andromaly: a behavioral malware detection framework for android devices', *J. Intell. Inf. Syst.*, 2012, **38**, (1), pp. 161–190

[17] Shabtai, A., Tenenboim-Chekina, L., Mimran, D., *et al.*: 'Mobile malware detection through analysis of deviations in application network behavior', *Comput. Secur.*, 2014, **43**, pp. 1–18

[18] Rozenberg, B., Gudes, E., Elovici, Y., *et al.*: 'A method for detecting unknown malicious executables'. Proc. of 10th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications (TustCom 2011), Shanghai, China, November 2011, pp. 190–196

[19] Gupta, V., Dharmaraja, S.: 'Semi-Markov modeling of dependability of VoIP network in the presence of resource degradation and security attacks', *Reliab. Eng. Syst. Saf.*, 2011, **96**, (12), pp. 1627–1636

[20] Kuang, G.C., Wang, X.F., Yin, L.R.: 'A fuzzy forecast method for network security situation based on Markov'. Proc. of IEEE Int. Conf. on Computer Science and Information Processing (CSIP), Shanxi, China, August 2012, pp. 785–789

[21] Rahman, M.: 'DroidMLN: a Markov logic network approach to detect android malware'. Proc. of 12th Int. Conf. on Machine Learning and Applications (ICMLA'13), Florida, USA, December 2013, pp. 166–169

[22] Xiao, X., Tian, X., Zhai, Q., *et al.*: 'A variable-length model for masquerade detection', *J. Syst. Softw.*, 2012, **85**, (11), pp. 2470–2478

[23] Zhou, Y., Jiang, X.: 'Dissecting android malware: characterization and evolution'. Proc. of IEEE Int. Conf. on Symp. Security and Privacy, San Francisco, USA, May 2012, pp. 95–109

[24] Sheldon, M.R.: '*Stochastic processes*' (John Wiley & Sons, Inc., 1996, 2nd edn.)

[25] Ronald, W.W.: '*Stochastic modeling and the theory of queues*' (Prentice-Hall, London, 1989)

[26] Lennartsson, J., Baxevani, A., Chen, D.: 'Modelling precipitation in Sweden using multiple step Markov chains and a composite model', *J. Hydrol.*, 2008, **363**, (1), pp. 42–59

[27] Haykin, S.S.: '*Neural networks and learning machines*' (Pearson Education, New Jersey, 2006, 3rd edn.)

[28] Basheer, I.A., Hajmeer, M.: 'Artificial neural networks: fundamentals, computing, design, and application', *J. Microbiol. Methods*, 2000, **43**, (1), pp. 3–31

[29] Hecht-Nielsen, R.: 'Theory of the backpropagation neural network'. Proc. of IEEE Int. Joint Conf. on Neural Networks (IJCNN), Washington, DC, USA, June 1989, pp. 593–605

[30] Zheng, M., Sun, M., Lui, J.: 'Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware'. Proc. of 12th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom 2013), Melbourne, Australia, July 2013, pp. 163–171

[31] Zhou, W., Zhou, Y., Jiang, X., *et al.*: 'Detecting repackaged smartphone applications in third-party android marketplaces'. Proc. of 2nd ACM Conf. on Data and Application Security and Privacy, New York, USA, 2012, pp. 317–326

[32] Aafer, Y., Du, W., Yin, H.: 'DroidAPIMiner: mining API-level features for robust malware detection in android'. Proc. of 9th Int. Conf. on Security and Privacy in Communication Networks (SecureComm2013), Sydney, Australia, September 2013, pp. 86–103

[33] Wei, T.E., Mao, C.H., Jeng, A.B., *et al.*: 'Android malware detection via a latent network behavior analysis'. Proc. of 11th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom 2012), Liverpool, UK, June 2012, pp. 1251–1258